PizzaLang: A Language for Hungry Developers

Ronik Bhaskar
The University of Chicago

Dawson Ren
Northwestern University

Abstract

Modern programming languages lack a clear focus on feeding their hungry programmers. As a first step, we introduce the *PizzaLang* programming language. With a straightforward type system, easy-to-read concrete syntax, and extensive features, *PizzaLang* outperforms state-of-the-art pizza programming languages. We rigorously prove powerful properties about *PizzaLang*, including strong normalization. Our implementation of a *PizzaLang* interpreter provides tangible results, demonstrating the effectiveness of this language.

1 Introduction

It's 11pm. You've been programming all day, and you forgot to eat dinner. With the project deadline approaching, it's not like you had time. You can't keep working on an empty stomach, but you can't step away from the keyboard either.

Pull yourself together. You're a programmer. You're a professional problem solver. You can get some food and keep writing code at the same time. All you need are the right tools.

Currently, these tools are APIs wrapped in APIs wrapped in dynamically-typed, object-oriented languages that don't care about your pizza. While they serve the necessary purpose, the sheer level of overhead makes actual use much more difficult than anticipated. Developers are stuck reading through pages of documentation, only to realize the pizza shop has closed, and their dinner plans will never be realized.

PizzaLang aims to provide better tools to hungry programmers. The language has one main purpose: constructing pizzas. Using clean structure, straightforward typing, and simple grammar, *PizzaLang* allows users to construct terms suitable for any pizza-related project, including ordering pizza.

2 Term Grammar

The term grammar of *PizzaLang* is as follows:

t ::= bp	*base pizza*
| left	*left half of pizza*
| right	*right half of pizza*
| all	*whole pizza*
| mushrooms	
| onions	
| green peppers	
| ...	*pizza toppings*
| add t t t	*add topping*
| remove t t	*remove topping*

This allows for the construction of the following values. All values are specified by v, following the style of Types and Programming Languages [1]. Pizza values are denoted by pv, topping values are denoted by tv, and location values are denoted by lv.

v ::= pv	*pizza values*
| tv	*topping values*
| lv	*location values*

tv ::= mushrooms | onions | green peppers | ...

lv ::= left
| right
| all

pv ::= bp
| add pv tv lv

Finally, this language has three types: Pizza, Topping, and Location.

$$\mathcal{T} ::= \text{Pizza} \mid \text{Topping} \mid \text{Location}$$

3 Evaluation Rules

The add and remove terms in *PizzaLang* function similarly to successor and predecessor respectively in Peano Arithmetic. The base pizza behaves like zero. The predecessor of zero is still zero, so removing a topping from a plain pizza results in a plain pizza. Wrapping add terms around a base pizza creates a larger pizza.

Unlike Peano Arithmetic, not all adds and removes are created equal. A remove around an add will only cause the two to cancel if they describe the same topping. Logically, removing mushrooms shouldn't also cause the green peppers to be removed from your pizza. Since adds can be nested, the remove term can descend through the adds, searching for the correct topping to remove.

Computation Rules

$$\frac{v_2 = v_4}{\text{remove } (\text{add } v_1\ v_2\ v_3)\ v_4 \to v_1}$$

$$\frac{v_2 \neq v_4}{\text{remove } (\text{add } v_1\ v_2\ v_3)\ v_4 \to \text{add } (\text{remove } v_1\ v_4)\ v_2\ v_3}$$

$$\frac{}{\text{remove bp } t_2 \to \text{bp}}$$

Congruence Rules

$$\frac{t_1 \to t_1'}{\text{add } t_1\ t_2\ t_3 \to \text{add } t_1'\ t_2\ t_3}$$

$$\frac{t_2 \to t_2'}{\text{add } v_1\ t_2\ t_3 \to \text{add } v_1\ t_2'\ t_3}$$

$$\frac{t_3 \to t_3'}{\text{add } v_1\ v_2\ t_3 \to \text{add } v_1\ v_2\ t_3'}$$

$$\frac{t_1 \to t_1'}{\text{remove } t_1\ t_2 \to \text{remove } t_1'\ t_2}$$

$$\frac{t_2 \to t_2'}{\text{remove } v_1\ t_2 \to \text{remove } v_1\ t_2'}$$

4 Type Rules

Since *PizzaLang* does not utilize variables, *PizzaLang* does not require a context Γ to store the types of variables. Instead, toppings are specified as fixed set of terms in the grammar, allowing independent developers to determine whether or not "pineapple" is valid syntax. Also, depending on the implementation of the interpreter/compiler, "mushrooms" is a well-typed program.

$$\frac{}{\text{bp} : \text{Pizza}}$$

$$\frac{}{\text{left} : \text{Location}}$$

$$\frac{}{\text{right} : \text{Location}}$$

$$\frac{}{\text{all} : \text{Location}}$$

$$\frac{}{\textit{topping name} : \text{Topping}}$$

$$\frac{t_1 : \text{Pizza} \qquad t_2 : \text{Topping} \qquad t_3 : \text{Location}}{\text{add } t_1\ t_2\ t_3 : \text{Pizza}}$$

$$\frac{t_1 : \text{Pizza} \qquad t_2 : \text{Topping}}{\text{remove } t_1\ t_2 : \text{Pizza}}$$

Given our type system, you may have noticed that the second, third, and fifth congruence rules are redundant, since terms of type Location or Topping are already values. These congruence rules future-proof the language for new terms of type Location or Topping that step to a value in one or more steps.

5 Strong Normalization

We claim that our language has strong normalization [1]. If a term in PizzaLang is well-typed, then it steps to a value in zero or more steps. The proof is by induction over the depth of the term. See Appendix A for the full, mechanized proof.

For far too long, Big Pizza has upheld too many barriers to getting your pizza. Pop-ups, redirects, ads, and confusing choices continue to plague the modern pizza consumer. As independent researchers, we hope to introduce robust systems for ordering pizza. This begins with the strong guarantee that your pizza order can produce a pizza.

Pizza-ordering languages that lack strong normalization risk infinite pizza loops, introducing topological paradoxes about how to apply sauce to a Möbius Pizza.

6 Concrete Syntax

The initial interpreter, written in Typed Racket, replicates a spoken-English pizza order with its syntax. It only specifies three different toppings, and they must be spelled as described. Furthermore, all programs must begin with "i'd like uh..." and end with "don't forget to bake it". We feel the first syntax requirement adds to the programming experience, mimicking common speech patterns for ordering food. The second requirement is entirely to annoy programmers who forget to add it.

t ::=	"i'd like uh..."	
	t	*main body*
	"don't forget to bake it"	
|	"pizza"	*base pizza*
|	t "with" t "on the" t	*add topping*
|	t "and" t "on the" t	*add topping*
|	t "actually hold the" t	*remove topping*
|	t "and" t	*remove topping*
|	"left half"	*location*
|	"right half"	*location*
|	"whole thing"	*location*
|	"mushrooms"	
|	"onions"	
|	"green peppers"	

Here is an example program:

```
i'd like uh...
pizza with mushrooms on the left half
and green peppers on the right half
and onions on the whole thing
actually
hold the onions and mushrooms
don't forget to bake it
```

The program evaluates to the following term:

```
add bp "green peppers" right
```

7 Further Applications

The most important values in the program are those of type Pizza. They can be read as a series of instructions, describing which toppings to add to a base pizza, in what order, and where on the pizza. To allow for further computations, an interpreter could use the sequence of toppings as computational instructions. For example, you may read the location of the topping as a left/right/stay move on a Turing tape, and the topping itself may tell you how to manipulate that square. Alternatively, you could feed the instructions to a program that draws pizzas.

Beyond computation, *PizzaLang* creates a context-free grammar for ordering pizza. Pizza delivery APIs have existed for years. Individual users can define what a base pizza means to them, and most pizza-delivery restaurants take orders as sequences of toppings to put on a pizza. The instructions may need to be adapted to fit the nuance of each pizza restaurant, but the logic is still clear.

Programmers of the world, open your editors. Let your stomachs be your guide. Write some code, and order some pizza.

References

[1] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2002.

Special thanks to Adam Shaw and Vincent St-Amour.

We acknowledge that this paper does not necessarily reflect the beliefs and ideas of our respective universities. We also acknowledge that the the rivalry between the universities is silly because one is clearly better than the other.

A Appendix

We claim PizzaLang has strong normalization. The following mechanized proof is written in Agda.

```
module pizza-lang-strong-normalization-proof where

open import PizzaLang.Theorems using (strong-norm-proof; WellTyped; steps-to-val)
open import PizzaLang.Theorems using (pizza-lang)

pl-strong-norm : Set
pl-strong-norm = ∀(x : WellTyped) → (steps-to-val x)

pl-strong-norm-proof : ∀(x : WellTyped) → (steps-to-val x)
pl-strong-norm-proof = (strong-norm-proof pizza-lang)
```